Learning Stochastic Logic Programs

Stephen Muggleton Department of Computer Science,

University of York,

York, YO1 5DD, United Kingdom.

Abstract **Stochastic Logic Programs** (SLPs) have been shown to be a generali- sation of Hidden Markov Models (HMMs), **stochastic** context-free gram- mars, and directed Bayes' nets. A **stochastic logic** program consists of a set of labelled clauses p:C where p is in the interval [0,1] and C is a first-order range-restricted definite clause. This paper summarises the syntax, distributional semantics and proof techniques for SLPs and then discusses how a standard Inductive **Logic** Programming (ILP) system, Progol, has been modified to support **learning** of SLPs. The resulting system 1) finds an SLP with uniform probability labels on each defini- tion and near-maximal Bayes posterior probability and then 2) alters the probability labels to further increase the posterior probability. Stage 1) is implemented within CProgol4.5, which differs from previous versions of Progol by allowing user-defined evaluation functions written in Prolog. It is shown that maximising the Bayesian posterior function involves finding SLPs with short derivations of the examples. Search pruning with the Bayesian evaluation function is carried out in the same way as in previous versions of CProgol. The system is demonstrated with worked examples involving the **learning** of probability distributions over sequences as well as the **learning** of simple forms of uncertain knowledge.

1 Introduction Representations of uncertain knowledge can be divided into a) procedural de- scriptions of sampling distributions (eg. **stochastic** grammars [4] and Hidden Markov Models (HMMs)) and b) declarative representations of uncertain state- ments (eg. probabilistic logics [2] and Relational Bayes' nets [3]). Stochas- tic **Logic Programs** (SLPs) [10] were introduced originally as a way of lifting **stochastic** grammars (type a representations) to the level of first-order **Logic Programs** (LPs). Later Cussens [1] showed that SLPs can be used to represent undirected Bayes' nets (type b representations). SLPs are presently used [9]

1

to define distributions for sampling within Inductive **Logic** Programming (ILP) [7].

Previous papers describing SLPs have concentrated on their procedural (sam- pling) interpretation. This paper first summarises the semantics and proof tech- niques for SLPs. The paper then describes a method for **learning** SLPs from examples and background knowledge.

The paper is organised as follows. Section 2 introduces standard definitions for LPs. The syntax, semantics and proof techniques for SLPs are given in Sec- tion 3. Incomplete SLPs are shown to have multiple consistent distributional models. Section 4 introduces a framework for **learning** SLPs and discusses is- sues involved with construction of the underlying LP as well as estimation of the probability labels. An overview of the ILP system Progol [6] is given in Section 5. Section 6 describes the mechanism which allows user-defined evalu- ation functions in Progol4.5 and derives the user-defined function for **learning** SLPs. Worked examples of **learning** SLPs are then given in Section 7. Section 8 concludes and discusses further work.

2 LPs The following summarises the standard syntax, semantics and proof techniques for LPs (see [5]).

2.1 Syntax of LPs A variable is denoted by an upper case letter followed by lower case letters and digits. Predicate and function symbols are denoted by a lower case letter followed by lower case letters and digits. A variable is a term, and a function symbol immediately followed by a bracketed n-tuple of terms is a term. In the case that n is zero the function symbol is a constant and is written without brackets. Thus f (g(X); h) is a term when f , g and h are function symbols, X is a variable and h is a constant. A predicate symbol immediately followed by a bracketted n-tuple of terms is called an atomic formula, or atom. The negation symbol is: :. Both a and :a are literals whenever a is an atom. In this case a is called a positive literal and :a is called a negative literal. A clause is a finite set of literals, and is treated as a universally quantified disjunction of those literals. A clause is said to be unit if it contains exactly one atom. A finite set of clauses is called a clausal theory and is treated as a conjunction of those clauses. Literals, clauses, clausal theories, True and False are all well-formed-formulas (wffs). A wff or a term is said to be ground whenever it contains no variables. A Horn clause is a clause containing at most one positive literal. A definite clause is a clause containing exactly one positive literal and is written as h b

1

; ::; b

n

where h is the positive literal, or head and the b

i

are negative literals, which

together constitute the body of the clause. A definite clause for which all the

2

variables in the head appear at least once in the body is called range-restricted. A non-definite Horn clause is called a goal and is written b

1

; ::; b

n

. A Horn

theory is a clausal theory containing only Horn clauses. A definite program is a clausal theory containing only definite clauses. A range-restricted definite program is a definite program in which all clauses are range-restricted.

2.2 Semantics of LPs Let ` = fv

1

=t

1

; :::; v

n

=t

n

g. ` is said to be a substitution when each v

i

is a variable

and each t

i

is a term, and for no distinct i and j is v

i

the same as v .

j

. Greek

lower-case letters are used to denote substitutions. ` is said to be ground when all t

i

are ground. Let E be a wff or a term and ` = fv

1

=t

1

; :::; v

n

=t

n

g be a

substitution. The instantiation of E by `, written E`, is formed by replacing every occurrence of v

i

in E by t

i

. E` is an instance of E. Clause C `-subsumes

clause D, or C _ D iff there exists a substitution theta such that C` ` D.

A first-order language L is a set of wffs which can be formed from a fixed and finite set of predicate symbols, function symbols and variables. A set of ground literals I is called an L-interpretation (or simply interpretation) in the case that it contains either a or :a for each ground atom a in L. Let M be an interpretation and C = h B be a definite clause in L. M is said to be an L-model (or simply model) of C iff for every ground instance h

0

B

0

of C

in L B

0

` M implies h

0

2 M . M is a model of Horn theory P whenever M

is a model of each clause in P . P is said to be satisfiable if it has at least one model and unsatisfiable otherwise. Suppose L is chosen to be the smallest first- order language involving at least one constant and the predicate and function symbols of Horn theory P . In this case an interpretation is called a Herbrand interpretation of P and the ground atomic subset of L is called the Herbrand Base of P . I is called a Herbrand model of Horn theory P when I is both Herbrand and a model of P . According to Herbrand's theorem P is satisfiable iff it has a Herbrand model. Let F and G be two wffs. We say that F entails G, or F j= G, iff every model of F is a model of G.

2.3 Proof for LPs An inference rule I = F ! G states that wff F can be rewritten by wff G. We say F `

I

G iff there exists a series of applications of I which transform F to G.

I is said to be sound iff for each F `

I

G always implies F j= G and complete

when F j= G always implies F `

I

G. I is said to be refutation complete if

I is complete with G restricted to False. The substitution ` is said to be the unifier of the atoms a and a

0

whenever a` = a

0

`. _ is the most general unifier

(mgu) of a and a

0

if and only if for all unifiers fl of a and a

0

there exists a

substitution AE such that (a_)AE = afl. The resolution inference rule is as follows. ((C n fag) [(D n f:a

0

g))` is said to be the resolvent of the clauses C and D

whenever C and D have no common variables, a 2 C, :a

0

2 D and ` is the

3

mgu of a and a

0

. Suppose P is a definite program and G is a goal. Resolution

is linear when D is restricted to clauses in P and C is either G or the resolvent of another linear resolution. The resolvent of such a linear resolution is another goal. Assuming the literals in clauses are ordered, a linear resolution is SLD when the literal chosen to resolve on is the first in C. An SLD refutation from P is a sequence of such SLD linear resolutions, which can be represented by D

P;G

= hG; C

1

; ::; C

n

i where each C

i

is in P and the last resolvent is the empty

clause (ie. False). The answer substitution is `

P;G

= `

1

`

2

::`

n

where each `

i

is the substitution corresponding with the resolution involving C

i

in D

P;G

. If

P is range-restricted then `

P;G

will be ground. SLD resolution is known to be

both sound and refutation complete for definite **programs**. Thus for a range- restricted definite program P and ground atom a it can be shown that P j= a by showing that P; a `

SLD

False. The Negation-by-Failure (NF) inference

rule says that P; a 6`

SLD

False implies P `

SLDNF

:a.

3 SLPs 3.1 Syntax of SLPs An SLP S is a set of labelled clauses p:C where p is a probability (ie. a number in the range [0; 1]) and C is a first-order range-restricted definite clause

1

. The

subset S

p

of clauses in S with predicate symbol p in the head is called the

definition of p. For each definition S

p

the sum of probability labels ss

p

must

be at most 1. S is said to be complete if ss

p

= 1 for each p and incomplete

otherwise. P (S) represents the definite program consisting of all the clauses in S, with labels removed.

Example 1 Unbiased coin. The following SLP is complete and represents a coin which comes up either heads or tails with probability 0.5.

S

1

=

ae

0:5 : coin(head)

0:5 : coin(tail)

oe

S

1

is a simple example of a sampling distribution

2

.

Example 2 Pet example. The following SLP is incomplete.

S

2

=

ae

0:3 : likes(X; Y) pet(Y; X); pet(Z; X);

cat(Y); mouse(Z)

oe

S

2

shows how statements of the form Pr(P (~x)jQ(~y)) = p can be encoded within

an SLP, in this case Pr(likes(X,Y)j. . .)) = 0:3.

1

Cussens [1] considers a less restricted definition of SLPs.

2

Section 7 provides a more complex sampling distribution a language by attaching prob-

ability labels to productions of a grammar. The grammar is encoded as a range-restricted definite program.

4

3.2 Proof for SLPs A **Stochastic** SLD (SSLD) refutation is a sequence D

S;G

= h1:G; p

1

:C

1

; :::; p

n

:C

n

i

in which G is a goal, each p

i

:C

i

2 S and D

P (S);G

= hG; C

1

; :::; C

n

i is an SLD

refutation from P (S). SSLD refutation represents the repeated application of the SSLD inference rule. This takes a goal p:G and a labelled clause q:C and produces the labelled goal pq:R, where R is the SLD resolvent of G and C. The answer probability of D

S;G

is Q(D

S;G

) =

Q

n

i=1

p

i

. The incomplete probability

of any ground atom a with respect to S is Q(ajS) =

P

D

S;(a)

Q(D

S;(a)

). We

can state this as S `

SSLD

Q(ajS) ^ P r(ajS) ^ 1, where P r(ajS) represents the

conditional probability of a given S.

Remark 3 Incomplete probabilities. If a is a ground atom with predicate symbol p and the definition S

p

in SLP S is incomplete then Q(ajS) ^ ss

p

.

Proof. Suppose the probability labels on clauses in S

p

are p

1

; ::; p

n

then Q(ajS) =

p

1

q

1

+ :: + p

n

q

n

where each q

i

is a sum of products for which 0 ^ q

i

^ 1. Thus

Q(ajS) ^ p

1

+ :: + p

n

= ss

p

.

3.3 Semantics of SLPs In this section we introduce the "normal" semantics of SLPs. Suppose L is a first-order language and D

p

is a probability distribution over the ground atoms of p in L. If I is a vector consisting of one such D

p

for every p in L then I is called

a distributional L-interpretation (or simply interpretation). If a 2 L is an atom with predicate symbol p and I is an interpretation then I(a) is the probability of a according to D

p

in I. Suppose L is chosen to be the smallest first-order

language involving at least one constant and the predicate and function symbols of Horn theory P (S). In this case an interpretation is called a distributional Herbrand interpretation of S (or simply Herbrand interpretation).

Definition 4 An interpretation M is a distributional L-model (or simply model) of SLP S iff Q(ajS) ^ M (a) for each ground atom a in L

3

.

Again if M is a model of S and M is Herbrand with respect to S then M is a distributional Herbrand model of S (or simply Herbrand model).

Example 5 Models.

S =

ae

0:5:p(X) q(X)

0:5:q(a)

oe

3

It might seem unreasonable to define semantics in terms of proofs in this way. However,

it should be noted that Q(ajS) represents a potentially infinite summation of the probabilities of individual SSLD derivations. This is analogous to defining the satisfiability of a first-order formula in terms of an infinite boolean expression derived from truth tables of the connectives

5

Q(p(a)jS) = 0:25 and Q(q(a)jS) = 0:5. L has predicate symbols p; q and con- stant a; b.

I

1

=

o/

f1:p(a); 0:p(b)g

f1:q(a); 0:q(b)

AE

I

1

is a model of S.

I

2

=

o/

f0:1:p(a); 0:9:p(b)g

f0:5:q(a); 0:5:q(b)

AE

I

2

is not a model of S.

Suppose S; T are SLPs. As usual we write S j= T iff every model of S is a model of T .

4 **Learning** SLPs 4.1 Bayes' function This section describes a framework for **learning** a complete SLP S from examples E based on maximising Bayesian posterior probability p(SjE). Below it is assumed that E consists of ground unit clauses. The posterior probability of S given E can be expressed using Bayes' theorem as follows.

p(SjE) =

p(S)p(EjS)

p(E)

(1)

p(S) represents a prior probability distribution over SLPs. If we suppose (as is normal) that the e

i

are chosen randomly and independently from some distri-

bution D over the instance space X then p(EjS) =

Q

m

i=1

p(e

i

jS). We assume

that p(e

i

jS) = Q(e

i

jS) (see Section 3.2). p(E) is a normalising constant. Since

the probabilities involved in the Bayes' function tend to be small it makes sense to re-express Equation 1 in information-theoretic terms by applying a negative log transformation as follows.

\Gamma log

2

p(SjE) = \Gamma log

2

p(S) \Gamma

m X

i=1

[log

2

p(e

i

jS)] + c (2)

Here \Gamma log

2

p(S) can be viewed as expressing the size (number of bits) of S.

The quantity \Gamma

P

m

i=1

[log

2

p(e

i

jS)] can be viewed as the sum of sizes (number of

bits) of the derivations of each e

i

from S. c is a constant representing log

2

p(E).

Note that this approach is similar to that described in [9], differing only in the definition of p(e

i

jS). The approach in [9] uses p(e

i

jS) to favour LP hypotheses

with low generality, while Equation 2 favours SLP hypotheses with a low mean derivation size. Surprisingly this makes the Bayes' function for **learning** SLPs appropriate for finding LPs which have low time-complexity with respect to the examples. For instance, this function would prefer an SLP whose underlying LP

6

represented quick-sort over one whose underlying LP represented insertion-sort since the mean proof lengths of the former would be lower than those of the latter.

4.2 Search strategy The previous subsection leaves open the question of how hypotheses are to be constructed and how search is to be ordered. The approach taken in this paper involves two stages.

1. LP construction. Choose an SLP S with uniform probability labels on

each definition and near maximal posterior probability with respect to E.

2. Parameter estimation. Vary the labels on S to increase the posterior

probability with respect to E.

Progol4.5 is used to implement the search in Stage 1. Stage 2 is implemented using an algorithm which assigns a label to each clause C in S according to the Laplace corrected relative frequency with which C is involved in proofs of the positive examples in E.

4.3 Limitations of strategy The overall strategy is sub-optimal in the following ways: a) the implementation of Stage 1 is approximate since it involves a greedy clause-by-clause construction of the SLPs, b) the implementation of Stage 2 is only optimal in the case that each positive example has a unique derivation.

5 Overview of Progol ILP systems take LPs representing background knowledge B and examples E and attempt to find the simplest consistent hypothesis H such that the following holds.

B ^ H j= E (3)

This section briefly describes the Mode Directed Inverse Entailment (MDIE) approach used in Progol [6]. Equation 3 is equivalent for all B, H and E to the following.

B ^ E j= H

Assuming that H and E are ground and that ? is the conjunction of ground literals which are true in all models of B ^ E we have the following.

B ^ E j= ?

7

Since H is true in every model of B ^ E it must contain a subset of the ground literals in ?. Hence

B ^ E j= ? j= H

and so for all H

H j= ? (4)

The set of solutions for H considered by Progol is restricted in a number of ways. Firstly, ? is assumed to contain only one positive literal and a finite number of negative literals. The set of negative literals in ? is determined by mode decla- rations (statements concerning the input/output nature of predicate arguments and their types) and user-defined restrictions on the depths of variable chains.

Progol uses a covering algorithm which repeatedly chooses an example e, forms an associated clause ? and searches for the clause which maximises the information compression within the following bounded sub-lattice.

2 _ H _ ? The hypothesised clause H is then added to the clause base and the examples covered by H are removed. The algorithm terminates when all examples have been covered. In the original version of Progol (CProgol4.1) [6] the search for each clause H involves maximising the `compression' function

f = (p \Gamma (c + n + h)) where p and n are the number of positive and negative examples covered by H, c is the number of literals in H, and h is the minimum number of additional literals required to complete the input/output variable chains in H (computed by considering variable chains in ?). In later versions of Progol the following function was used instead to reduce the degree of greediness in the search.

f =

m

p

(p \Gamma (c + n + h)) (5)

This function estimates the overall global compression expected of the final hypothesised set of clauses, extrapolated from local coverage and size properties of the clause under construction. A hypothesised clause H is pruned, together with all its more specific refinements, if either

1 \Gamma

c

p

^ 0 (6)

or there exists a previously evaluated clause H

0

such that H

0

is an accept-

able solution (covers below the noise threshold of negative examples and the input/output variable chains are complete) and

1 \Gamma

c

p

^ 1 \Gamma

c

0

+ n

0

+ h

0

p

0

(7)

where p; c are associated with H and p

0

; n

0

; c

0

; h

0

are associated with H

0

.

8

Variable Built-in User-defined p pos cover(P1) user pos cover(P2) n neg cover(N1) user neg cover(N2) c hyp size(C1) user hyp size(C2) h hyp rem(H1) user hyp rem(H2)

Figure 1: Built-in and user defined predicates for some of the variables from Equation 5.

6 User-defined evaluation in Progol4.5 User-defined evaluation functions in Progol4.5 are implemented by allowing re- definition in Prolog of p, n and c from Equation 5. Figure 1 shows the con- vention for names used in Progol4.5 for the built-in and user-defined functions for these variables. Though this approach to allowing definition of the evalu- ation function is indirect, it means that the general criteria used in Progol for pruning the search (see Inequalities 6 and 7) can be applied unaltered as long as user pos cover and user neg cover monotonically decrease and user hyp size monotonically increases with downward refinement (addition of body literals) to the hypothesised clause. For **learning** SLPs these functions are derived below.

Equation 2 can be rewritten in terms of an information function I as

I(SjE) = I(S) \Gamma

m X

i=1

I(e

i

jS) + c (8)

where I(x) = \Gamma log

2

x. The degree of compression achieved by an hypothesis is

computed by subtracting I(SjE) from I(S

0

= EjE), the posterior information

of the hypothesis consisting of returning ungeneralised examples.

I(S

0

= EjE) = I(E) + I(EjS

0

= E) + c

= m + mlog

2

m + c

= m(1 + log

2

m) + c (9)

The compression induced by S with respect to E is now simply the difference between Equations 9 and 8, which is as follows.

m(1 + log

2

m) \Gamma I(S) +

m X

i=1

I(e

i

jS)

=

m

p

(p(1 + log

2

m) \Gamma I(H) +

p X

j=1

I(e

j

jH)) (10)

9

Examples class(dog,mammal).

class(trout,fish). : : : Background has covering(dog,hair). knowledge : : :

has legs(dolphin,0). : : :

Figure 2: Examples and background knowledge for animal taxonomy. In Equation 10 extrapolation is made from the p positive examples covered by hypothesised clause H. Comparing Equations 5 and 10 the user-defined functions of Figure 1 are as follows (p; n; c; h represent built-in functions and p

0

; n

0

; c

0

; h

0

represent their user-defined counter-parts).

p

0

= p(1 + log

2

m) (11)

n

0

=

m X

j=1

I(e

j

jH) + n

c

0

= c

h

0

= h

7 Worked examples The source code of Progol4.5 together with the input files for the following worked examples can be obtained from ftp://ftp.cs.york.ac.uk/pub/ML GROUP/progol4.5/ .

7.1 Animal taxonomy Figure 2 shows the examples and background knowledge for an example set which involves **learning** taxonomic descriptions of animals. Following Stage 1 (Section 4.2) the SLP constructed has uniform probability labels as follows

4

0.200: class(A,reptile) :-

has.legs(A,4), has.eggs(A). 0.200: class(A,mammal) :- has.milk(A). 0.200: class(A,fish) :- has.gills(A). 0.200: class(A,reptile) :-

4

For this example and the next the value of p

0

(Equation 11) was increased by a factor of

4 to achieve positive compression

10

Examples s([the,man,walks,the,dog],[]).

s([the,dog,walks,to,the,man],[]). : : : Background np(S1,S2) :- det(S1,S3), noun(S3,S2). knowledge : : :

noun([manjS],S). : : :

Figure 3: Examples and background knowledge for Simple English Grammar.

has.legs(A,0), habitat(A,land). 0.200: class(A,bird) :-

has.covering(A,feathers).

Following Stage 2 the labels are altered as follows to reflect the distribution of class types within the training data.

0.238: class(A,reptile) :-

has.legs(A,4), has.eggs(A). 0.238: class(A,mammal) :- has.milk(A). 0.238: class(A,fish) :- has.gills(A). 0.095: class(A,reptile) :-

has.legs(A,0), habitat(A,land). 0.190: class(A,bird) :-

has.covering(A,feathers).

7.2 Simple English grammar Figure 3 shows the examples and background knowledge for an example set which involves **learning** a simple English grammar. Following Stage 2 the learned SLP is as follows.

0.438: s(A,B) :- np(A,C), vp(C,D),

np(D,B). 0.562: s(A,B) :- np(A,C), verb(C,D),

np(D,E), prep(E,F), np(F,B).

8 Conclusion This paper describes a method for **learning** SLPs from examples and background knowledge. The method is based on an approximate Bayes MAP (Maximum A Posterior probability) algorithm. The implementation within Progol4.5 is

11

eAEcient and produces meaningful solutions on simple domains. However, as pointed out in Section 4.3 the method does not find optimal solutions.

The author views the method described as a first attempt at a hard problem. It is believed that improvements to the search strategy can be made. This is an interesting topic for further research.

The author believes that **learning** of SLPs is of potential interest in all do- mains in which ILP has had success [7]. In these domains it is believed that SLPs would the advantage over LPs of producing predictions with attached de- grees of certainty. In the case of multiple predictions, the probability labels would allow for relative ranking. This is of particular importance for Natural Language domains, though would also have general application in Bioinformat- ics [8].

Acknowledgements The author would like to thank Wray Buntine, David Page, Koichi Furukawa and James Cussens for discussions on the topic of **Stochastic Logic** Programming. Many thanks are due to my wife, Thirza and daughter Clare for the support and happiness they give me. This work was supported partly by the Esprit RTD project "ALADIN' (project 28623), EPSRC grant "Closed Loop Machine **Learning**", BBSRC/EPSRC grant "Protein structure prediction - development and benchmarking of machine **learning** algorithms" and EPSRC ROPA grant "Machine **Learning** of Natural Language in a Computational **Logic** Framework".

References

[1] J. Cussens. Loglinear models for first-order probabilistic reasoning. In

Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence, pages 126-133, San Francisco, 1999. Kaufmann.

[2] R. Fagin and J. Halpern. Uncertainty, belief and probability. In Proceedings

of IJCAI-89, San Mateo, CA, 1989. Morgan Kauffman.

[3] M. Jaeger. Relational bayesian networks. In Proceedings of the Thirteenth

Annual Conference on Uncertainty in Artificial Intelligence, San Francisco, CA, 1997. Kaufmann.

[4] K. Lari and S. J. Young. The estimation of **stochastic** context-free gram-

mars using the inside-outside algorithm. Computer Speech and Language, 4:35-56, 1990.

[5] J.W. Lloyd. Foundations of **Logic** Programming. Springer-Verlag, Berlin,

1987. Second edition.

12

[6] S. Muggleton. Inverse entailment and Progol. New Generation Computing, 13:245-286, 1995.

[7] S. Muggleton. Inductive **logic** programming: issues, results and the LLL challenge. Artificial Intelligence, 114(1-2):283-296, December 1999.

[8] S. Muggleton. Scientific knowledge discovery using inductive **logic** programming. Communications of the ACM, 42(11):42-46, November 1999.

[9] S. Muggleton. **Learning** from positive data. Machine **Learning**, 2000. Accepted subject to revision.

[10] S.H. Muggleton. **Stochastic logic programs**. In L. de Raedt, editor, Advances in Inductive **Logic** Programming, pages 254-264. IOS Press, 1996.

Stochastic Logic Programs*

Stephen Muggleton
Oxford University Computing Laboratory,
Parks Road,
Oxford, OX1 3QD,
United Kingdom.

Abstract

One way to represent a machine learning algorithm's *bias* over the hypothesis and instance space is as a pair of probability distributions. This approach has been taken both within Bayesian learning schemes and the framework of U-learnability. However, it is not obvious how an Inductive Logic Programming (ILP) system should best be provided with a probability distribution. This paper extends the results of a previous paper by the author which introduced *stochastic logic programs* as a means of providing a structured definition of such a probability distribution. Stochastic logic programs are a generalisation of stochastic grammars. A stochastic logic program consists of a set of labelled clauses $p : C$ where p is from the interval $[0, 1]$ and C is a range-restricted definite clause. A stochastic logic program P has a distributional semantics, that is one which assigns a probability distribution to the atoms of each predicate in the Herbrand base of the clauses in P. These probabilities are assigned to atoms according to an SLD-resolution strategy which employs a stochastic selection rule. It is shown that the probabilities can be computed directly for *fail-free* logic programs and by normalisation for arbitrary logic programs. The stochastic proof strategy can be used to provide three distinct functions: 1) a method of sampling from the Herbrand base which can be used to provide selected targets or example sets for ILP experiments, 2) a measure of the information content of examples or hypotheses; this can be used to guide the search in an ILP system and 3) a simple method for conditioning a given stochastic logic program on samples of data. Functions 1) and 3) are used to measure the generality of hypotheses in the ILP system Progol4.2. This supports an implementation of a Bayesian technique for learning from positive examples only.

*This paper is an extension of a paper with the same title which appeared in [12]

1 Introduction

The integration of logic and probability theory has been the subject of many studies [1, 3, 9, 5, 6, 17, 20]. It has also been investigated within logic programming [16, 19]. Recently the motivation for many such studies has been the development of formalisms for rule-based expert systems which employ uncertain reasoning.

By contrast, the motivation for the present paper comes from machine learning. One way to represent a machine learning algorithm's *bias* over the hypothesis and instance space is as a pair of probability distributions. This approach has been taken in various ways within the frameworks of PAC-learnability [21], Bayesian learning [2, 7] and U-learnability [11, 14]. However, it is not obvious how a machine learning algorithm, in particular an Inductive Logic Programming (ILP) system [10, 15] should best be provided with a probability distribution over a set of logical formulae. This paper proposes *stochastic logic programs* (SLPs) as a means of providing a structured definition of such a probability distribution. SLPs are a generalisation of stochastic grammars [8]. Although they have a distributional semantics, SLPs' relationship to Probabilistic Logic Programs [16] and BS-programs [19] is unclear.

This paper is organised as follows. In Section 2 the formal framework for U-learnability is introduced. Stochastic grammars are then defined and described in Section 3. Section 4 introduces stochastic logic programs as a generalisation of stochastic grammars. Section 5 describes a Prolog implementation of stochastic logic programs. A discussion of research issues and applications of stochastic logic programs concludes the paper in Section 6.

2 U-learnability

The following is a variant of the U-learnability framework presented in [11, 14]. The teacher starts by choosing distributions $D_{\mathcal{H}}$ and D_X from the family of distributions $\mathcal{D}_{\mathcal{H}}$ and $\mathcal{D}_X$ over concept descriptions $\mathcal{H}$ (wffs with associated bounds for time taken to test entailment) and instances X (ground wffs) respectively. The teacher uses $D_{\mathcal{H}}$ and D_X to carry out an infinite series of teaching sessions. In each session a target theory T is chosen from $D_{\mathcal{H}}$. Each T is used to provide labels from $\{\blacksquare, \square\}$ (True, False) for a set of instances randomly chosen according to distribution D_X. The teacher labels each instance x_i in the series $\langle x_1, .., x_m \rangle$ with $\blacksquare$ if $T \models x_i$ and $\square$ otherwise. An hypothesis $H \in \mathcal{H}$ is said to explain a set of examples E whenever it both entails and is consistent with E. On the basis of the series of labelled instances $\langle e_1, e_2, .., e_m \rangle$, a Turing machine learner L produces a sequence of hypotheses $\langle H_1, H_2, ..H_m \rangle$ such that $H_i \in \mathcal{H}$ explains $\{e_1, .., e_i\}$. H_i must be suggested by L in expected time bounded by a fixed polynomial function of i. The teacher stops a session once the learner suggests hypothesis H_m with expected error less than ϵ for the label of any x_{m+1} chosen

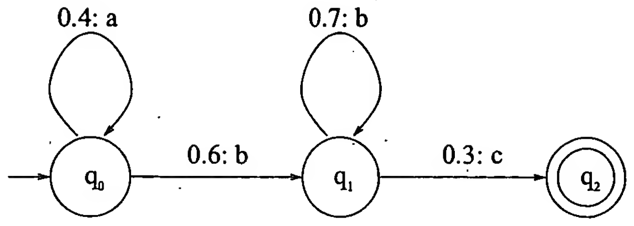

Figure 1: Stochastic automaton.

randomly from D_X. $\langle D_{\mathcal{H}}, D_X \rangle$ is said to be U-learnable if and only if there exists a Turing machine learner L such that for any choice of δ and ϵ $(0 < \delta, \epsilon \leq 1)$ with probability at least $(1 - \delta)$ in any of the sessions m is less than a fixed polynomial function of $\frac{1}{\delta}$ and $\frac{1}{\epsilon}$.

In [14] positive results were given for the U-learnability of time-bounded logic programs. For these proofs it was assumed that a logic program P could be chosen by the teacher according to a distribution $D_{\mathcal{H}}$ and that instances could be chosen from the Herbrand base of P according to a distribution D_X. Clearly both $D_{\mathcal{H}}$ and D_X are functions over countably infinite domains. Stochastic grammars provide one approach to defining a probability distribution over a countably infinite set.

3 Stochastic grammars

3.1 Stochastic automata

Stochastic automata, otherwise called Hidden Markov Models [18], have found many applications in speech recognition. An example is shown in Figure 1. Stochastic automata are defined by a 5-tuple $A = \langle Q, \Sigma, q_0, F, \delta \rangle$. Q is a set of states. Σ is an alphabet of symbols. q_0 is the initial state and $F \subseteq Q$ ($F = \{q_2\}$ in Figure 1) is the set of final states. $\delta : (Q \setminus F) \times \Sigma \rightarrow Q \times [0,1]$ is a stochastic transition function which associates probabilities with labelled transitions between states. The sum of probabilities associated with transitions from any state $q \in (Q \setminus F)$ is 1.

In the following λ represents the empty string. The transition function $\delta^* : (Q \setminus F) \times \Sigma^* \rightarrow Q \times [0,1]$ is defined as follows. $\delta^*(q, \lambda) = \langle q, 1 \rangle$. $\delta^*(q, au) = \langle q_{au}, p_a p_u \rangle$ if and only if $\delta(q, a) = \langle q_a, p_a \rangle$ and $\delta^*(q_a, u) = \langle q_{au}, p_u \rangle$. The probability of u being accepted from state q in A is defined as follows. $Pr(u|q, A) = p$ if $\delta^*(q, u) = \langle q', p \rangle$ and $q' \in F$. $Pr(u|q, A) = 0$ otherwise.

Theorem 1 Probability of a string being accepted from a particular state. *Let $A = \langle Q, \Sigma, q_0, F, \delta \rangle$ be a stochastic automaton. For any $q \in Q$ the* *following holds.*

$$\sum_{u \in \Sigma^*} Pr(u|q, A) = 1.$$

Proof. *Suppose the theorem is false. Either $q \in F$ or $q \notin F$. Suppose $q \in F$. Then by the definition of stochastic automata q has no outgoing transitions. Therefore by definition $Pr(u|q, A)$ is 1 for $u = \lambda$ and 0 otherwise, which is in accordance with the theorem. Therefore suppose $q \notin F$. Suppose in state q the transitions are $\delta(q, a_1) = \langle q, p_1 \rangle, \ldots, \delta(q, a_n) = \langle q, p_n \rangle$. Then each string u is accepted in proportions $p_1, \ldots, p_n$. according to its first symbol. That is to say, $\sum_{u \in \Sigma^*} Pr(u|q, A) = p_1 + .. + p_n$. But according to the definition of δ, $p_1 + \ldots + p_n = 1$, which means $\sum_{u \in \Sigma^*} Pr(u|q, A) = 1$. This contradicts the assumption and completes the proof.* □

If the probability of u being accepted by A is now defined as $Pr(u|A) \doteq Pr(u|q_0, A)$ then the following corollary shows that A defines a probability distribution over Σ^*.

Corollary 2 Stochastic automata represent probability distributions. *Given stochastic automaton A.*

$$\sum_{u \in \Sigma^*} Pr(u|A) = 1.$$

Proof. *Special case of Theorem 1 when $q = q_0$.* □

The following example illustrates the calculation of probabilities of strings.

Example 3 Probabilities associated with strings. *For the automaton A in Figure 1 we have $Pr(abbc|A) = 0.4 \times 0.6 \times 0.7 \times 0.3 = 0.0504$. $Pr(abac|A) = 0$.*

A can also be viewed as expressing a probability distribution over the language $L(A) = \{u : \delta^*(q_0, u) = \langle q, p \rangle \text{ and } q \in F\}$. The following theorem places bounds on the probability of individual strings in $L(A)$. The notation $|u|$ is used to express the length of string u.

Theorem 4 Probability bounds. *Let $A = \langle Q, \Sigma, q_0, F, \delta \rangle$ be a stochastic automaton and let p_{min}, p_{max} be respectively the minimum and maximum probabilities of any transition in A. Let $u \in L(A)$ be a string.*

$$p_{min}^{|u|} \leq Pr(u|A) \leq p_{max}^{|u|}.$$

Proof. *$Pr(u|A) = \prod_{i=1}^{|u|} p_i$, where p_i is the probability associated with the ith transition in A accepting u. Clearly each p_i is bounded below by p_{min} and above by p_{max}, and thus $p_{min}^{|u|} \leq Pr(u|A) \leq p_{max}^{|u|}$.* □

This theorem shows that a) all strings in $L(A)$ have non-zero probability and b) stochastic automata express probability distributions that decrease exponentially in the length of strings in $L(A)$.

$$\begin{aligned}0.4 &: q_0 \rightarrow aq_0\\ 0.6 &: q_0 \rightarrow bq_1\\[1em] 0.7 &: q_1 \rightarrow bq_1\\ 0.3 &: q_1 \rightarrow cq_2\\[1em] 1.0 &: q_2 \rightarrow \lambda\end{aligned}$$

Figure 2: Labelled production rule representation of stochastic automaton.

3.2 Labelled productions

Stochastic automata can be equivalently represented as a set of labelled production rules. Each state in the automaton is represented by a non-terminal symbol and each δ transition $\langle q, a\rangle \rightarrow \langle q', p\rangle$ is represented by a production rule of the form $p : q \rightarrow aq'$. Figure 2 is the set of labelled production rules corresponding to the stochastic automaton of Figure 1. Strings can now be generated from this stochastic grammar by starting with the string q_0 and progressively choosing productions to rewrite the leftmost non-terminal randomly in proportion to their probability labels. The process terminates once the string contains no non-terminals. The probability of the generated string is the product of the labels of rewrite rules used.

3.3 Stochastic context-free grammars

Stochastic context-free grammars [8] can be treated in the same way as the labelled productions of the last section. However, the following differences exist between the regular and context-free cases.

- To allow for the expression of context-free grammars the left-hand sides of the production rules are allowed to consist of arbitrary strings of terminals and non-terminals.
- Since context-free grammars can have more than one derivation of a particular string u, the probability of u is the sum of the probabilities of the individual derivations of u.
- The analogue of Theorem 4 holds only in relation to the length of the derivation, not the length of the generated string.

Example 5 The language $a^n b^n$. *Figure 3 shows a stochastic context-free grammar G expressed over the language $a^n b^n$. The probabilities of generated strings are as follows.* $Pr(\lambda|G) = 0.5$, $Pr(ab|G) = 0.25$, $Pr(aabb|G) = 0.125$.

$$0.5 : S \to \lambda$$
$$0.5 : S \to aSb$$

Figure 3: Stochastic context free grammar

$$0.5 : coin(0) \leftarrow$$
$$0.5 : coin(1) \leftarrow$$

Figure 4: Simple SLP

4 Stochastic logic programs

Every context-free grammar can be expressed as a definite clause grammar [4]. For this reason the generalisation of stochastic context-free grammars to stochastic logic programs (SLPs) is reasonably straightforward. First, a definite clause C is defined in the standard way as having the following form.

$$A \leftarrow B_1, \ldots, B_n$$

where the atom A is the head of the clause and $B_1, \ldots, B_n$ is the body of the clause. C is said to be range-restricted if and only if every variable in the head of C is found in the body of C. A *stochastic clause* is a pair $p : C$ where p is in the interval $[0, 1]$ and C is a range-restricted clause. A set of stochastic clauses P is called a *stochastic logic program* if and only if for each predicate symbol q in P the probability labels for all clauses with q in the head sum to 1.

Example 6 Coin example. *Figure 4 shows a simple SLP which mimics the action of a fair coin. The probability of the coin coming up either head-side up* (0) *or tail-side up* (1) *is* 0.5.

4.1 Stochastic SLD-refutations

For SLPs the stochastic refutation of a goal is analogous to the stochastic generation of a string from a set of labelled production rules. Suppose that P is an SLP. Then $n(P)$ will be used to express the logic program formed by dropping all the probability labels from clauses in P. A stochastic SLD procedure will be used to define a probability distribution over the Herbrand base of $n(P)$. The stochastic SLD-derivation of atom a is as follows. Suppose $\leftarrow g$ is a unit goal with the same predicate symbol as a, no function symbols and distinct variables. Next suppose that there exists an SLD-refutation of $\leftarrow g$ with answer substitution θ such that $g\theta = a$. Since all clauses in $n(P)$ are range-restricted, θ is necessarily a ground substitution. The probability of each clause selection in the refutation is as follows. Suppose the first atom in the subgoal $\leftarrow g'$ can unify with the heads of stochastic clauses $p_1 : C_1, \ldots, p_n : C_n$, and stochastic clause $p_i : C_i$ is chosen in the refutation. Then the probability of this choice is $\frac{p_i}{p_1+\ldots+p_n}$. The probability of the derivation of a is the product of the probability of the choices in the refutation. As with stochastic context-free grammars, the probability of a is then the sum of the probabilities of the derivations of a.

This stochastic SLD-strategy corresponds to a distributional semantics [19] for P. That is, each atom a in the success set of $n(P)$ is assigned a non-zero probability (due to the completeness of SLD-derivation). For each predicate symbol q the probabilities of atoms in the success set of $n(P)$ corresponding to q sum to 1 (the proof of this is analogous to Theorem 1).

4.2 Fail-free SLPs

It should be noted that the definition of SLPs in the previous section has problems. For instance, consider the case in which P has an empty success set. In this case the probability distribution is not well defined since it does not sum to 1. A less extreme case occurs when at least some derivations exist, though other derivations reach a deadend in which the goal $\leftarrow g'$ cannot unify with the heads of any clauses. In this case the probabilities of individual atoms must be normalised by multiplying each derivation probability by the reciprocal of the sum of all such probabilities.

Fail-free logic programs avoid this issue, since no selection choice leads to enforced backtracking. Fail-free clauses, logic programs and SLPs are defined as follows[1].

Definition 7 Fail-free clause. *A clause $h \leftarrow B$ is said to be fail-free if and only if B contains no function symbols and each variable occurs at most once in B.*

Definition 8 Fail-free logic programs. *A logic program P is fail-free if and only if each clause in P is definite and fail-free, and each predicate symbol in the body of each clause in P occurs in the head of at least one clause in P.*

Definition 9 Fail-free stochastic logic programs. *A stochastic logic program P is fail-free if and only if $n(P)$ is a fail-free logic program.*

We now show fail-free logic programs avoid forced backtracking.

Theorem 10 Non-backtracking of fail-free logic programs. *Let P be a fail-free logic program and $\leftarrow g$ be a fail-free goal containing only predicate symbols found in P. Irrespective of the choices made in the stchastic SLD-derivation of goal $\leftarrow g$ there will be no subgoal $\leftarrow g'$ which fails to resolve with*

[1]Although the following constraints are sufficient to avoid forced backtracking, it is not clear whether they are also necessary for definition of this class.

$$\begin{aligned}&0.5 : nate(0) \leftarrow\\&0.5 : nate(s(N)) \leftarrow nate(N)\end{aligned}$$

Figure 5: Exponential distribution over natural numbers

the definition of any predicate in g'.
Proof. *Assume the theorem is false. First note that the resolvent of any fail-free goal and a fail-free definite clause is itself fail-free and that the unification involved is one-sided. However, there must exist an intermediate goal $\leftarrow g'$ with substitution θ' which fails to resolve with the predicate definition for a predicate p in P. But since every predicate symbol in $\leftarrow g$ has a definition in P and each predicate symbol in the body of each clause in P has a definition in P it must be that $\leftarrow g'\theta'$ fails to unify with any of the clauses in the definition of p. However, since all unifications are one-sided, θ' will contain no substitutions for the variables in $\leftarrow g'$, and thus the first atom of $\leftarrow g'\theta'$ will have no function symbols and distinct variables, and thus must be able to resolve with all clauses of the corresponding definition. This contradicts the assumption and completes the proof.* □

Note that the class of fail-free logic programs includes all normal unary definitions of types (such as *list/1* or *natural/1*) as well as the standard recursive definitions of predicates such as *member/2* and *append/3*.

4.3 Polynomial distributions

It is reasonable to ask whether Theorem 4 extends in some form to SLPs. The distributions described in [14] include both those that decay exponentially over the length of formulae and those that decay polynomially. SLPs can easily be used to describe an exponential decay distribution over the natural numbers as follows.

Example 11 Exponential distribution. *Figure 5 shows a recursive SLP P which describes an exponential distribution over the natural numbers expressed in Peano arithmetic form. The probabilities of atoms are as follows. $Pr(nate(0)|P) = 0.5$, $Pr(nate(s(0))|P) = 0.25$ and $Pr(nate(s(s(0)))|P) = 0.125$. In general $Pr(nate(N)|P) = 2^{-N-1}$.*

However, SLPs can also be used to define a polynomially decaying distribution over the natural numbers as follows.

Example 12 Polynomial distribution. *Figure 6 shows a recursive SLP P which describes a polynomial distribution over the natural numbers expressed in reverse binary form. Numbers are constructed by first choosing the length of*

$$1.0 : natp(N) \leftarrow nate(U), bin(U, N)$$

$$0.5 : bin(0, [1]) \leftarrow$$
$$0.5 : bin(s(U), [C|N]) \leftarrow coin(C), bin(U, N)$$

Figure 6: Polynomial distribution over natural numbers

the binary representation and then filling out the binary expression by repeated tossing of a fair coin (see Figure 4). Since the probability of choosing a number N *of length* $log_2(N)$ *is roughly* $2^{-log_2(N)}$ *and there are* $2^{log_2(N)}$ *such numbers, each with equal probability,* $Pr(natp(N)|P) \approx 2^{-2log_2(N)} = N^{-2}$.

5 A Prolog implementation

Stochastic logic programs can be used to provide three distinct functions.

1. **A sampler.** A method of sampling from the Herbrand base which can be used to provide selected targets or example sets for ILP experiments, Sampling targets requires an SLP that implements a grammar describing the hypothesis space.

2. **Information content.** A measure of the information content of examples or hypotheses. The information content of atom a relative to SLP P is taken here as simply $-log_2(Pr(a|P))$. This could be used to help guide the search in an ILP system.

3. **A conditioner.** A simple method for conditioning a given stochastic logic program on samples of data.

Prolog implementations of these three functions are described in the following sections.

5.1 The predicate *sample/1*

This section describes a Prolog interpreter for SLPs. Figure 7 shows the code for the top-level interpreter, which is similar to a standard 'proves' interpreter for Prolog. Note that backtracking is disabled by the use of cuts. The reason for this is that each atom should be sampled independently of its predecessors. The third clause uses the predicate *clause/3*, the third argument of which is a unique number associated with the returned clause. Though this predicate is non-standard it is relatively straightforward to implement. (It is implemented as a primitive in CProgol4.2 which can be obtained by anonymous ftp from ftp.comlab.ox.ac.uk in directory pub/Packages/ILP/progol4.2).

```
sample((Goal1,Goal2)) :-
        !, sample(Goal1), sample(Goal2). % Conjunction
sample(Goal) :-
        not(clause(Goal,_)), !, Goal.    % System predicate
sample(Head) :-
        bagof([Head,Body,N],clause(Head,Body,N),Bag),
        random_clause(Head,Body,Bag),    % Random choice
        !, sample(Body).                 % User predicate
```

Figure 7: The predicate sample/1

```
random_clause(Head,Body,Bag) :-
        Rand is random,               % 0-1 random number
        choose(Bag,Head,Body,0,Rand,Sum).

choose([],_,_,_,_,0).
choose([[Head,Body,N]|Bag],Head1,Body1,SoFar,Rand,Rest) :-
        label(N,P), SoFar1 is SoFar+P,
        choose(Bag,Head1,Body1,SoFar1,Rand,Rest1),
        Rest is Rest1+P,
        ((var(Body1), P1 is SoFar/(SoFar+Rest), Rand>=P1,
                Head1=Head, Body1=Body);
        true).
```

Figure 8: The predicates random_clause/3 and choose/6

The predicate *random_clause/3* and its sub-predicate *choose/6* are shown in Figure 8. In *choose/6* the probability label of the clause is extracted using the predicate *label/2*, which is also implemented as a primitive in CProgol4.2 (see above). Note that since *P1* is simply a ratio, it is immaterial whether the labels are themselves in the interval $[0, 1]$ or simply arbitrary positive reals. CProgol4.2 by default assigns all clauses a label value of 1. CProgol4.2 also contains alterations to a standard Prolog interpreter which allow efficient sampling of stochastic logic programs using the built-in predicate *sample/3*.

5.2 Information content

Predicates for computing the information content of atoms are shown in Figures 9 and 10. For simplicity it is assumed here that each atom has at most one proof, and that each unification is deterministically chosen given the substitution so far. The predicate *info/3* again acts like a 'proves' interpreter which computes the

```
info(Goal,Bits) :-
        functor(Goal,F,N),
        functor(GGoal,F,N),
        info(GGoal,Goal,Bits1),
        Bits is Bits1.

info((GGoal1,GGoal2),(SGoal1,SGoal2),Bits1+Bits2) :- !,
        info(GGoal1,SGoal1,Bits1),
        info(GGoal2,SGoal2,Bits2).  % Conjunction
info(GHead,SHead,Bits1+Bits2) :-
        bagof([GBody,GN],clause(GHead,GBody,GN),GBag),
        clause(SHead,SBody,SN),
        info_choice(GBag,SN,GBody,0,_,Bits1),
        !, info(GBody,SBody,Bits2). % User predicate
info(Goal,Goal,0) :-
        not(clause(Goal,_)), Goal.  % System predicate
```

Figure 9: The predicates *info/2* and *info/3*

```
info_choice([],_,_,_,0,_).
info_choice([[Body,N]|T],N,Body,SoFar,Rest,Bits) :-
        !, info_choice(T,N,_,SoFar,Rest,_),
        label(N,P), Bits is -log(P/(SoFar+P+Rest))/log(2).
info_choice([[_,N]|T],M,Body,SoFar,Rest+P,Bits) :-
        label(N,P), info_choice(T,M,Body,SoFar+P,Rest,Bits).
```

Figure 10: The predicate *info_choice/6*

probabilities of the choices of a given ground goal (Goal) relative to a general form of the same goal (GGoal). The predicate *info_choice/6* acts much like *choose/6* in Figure 8 except that it computes the negative log probability of the choice.

5.3 The predicate *condition/1*

Predicate *condition/1* in Figure 11 uses *label/1* to condition an SLP according to a given ground goal. The built-in predicate *label/1* in CProgol4.2 simply increments an integer label associated with each clause.

The conditioner can be used to learn the parameters of a given distribution from a given set of ground atomic clauses.

```
condition((Goal1,Goal2)) :-
        !, condition(Goal1), condition(Goal2). % Conjunction
condition(Goal) :-
        not(clause(Goal,_)), !, Goal.          % System predicate
condition(Head) :-
        clause(Head,Body,N), label(N),
        !, condition(Body).                    % User predicate
```

Figure 11: Predicate *condition/1*

6 Discussion

This paper introduces stochastic logic programs as a structural description of a learning system's biases over the hypothesis and instance spaces. Since SLPs seem a simple and natural extension of logic programs it is hoped that they might find further applications within logic programming. Possible areas for application include robotics, planning and natural language.

SLPs have been applied in the problem of learning from positive examples only [13]. This required the implementation of the following function which defines the generality of an hypothesis.

$$g(H) = \sum_{x \in H} D_X(x).$$

The generality is thus the sum of the probability of all instances of hypothesis H. Clearly such a sum can be infinite. However, if a large enough sample is generated from D_X (implemented as an SLP) then the proportion of the sample entailed by H gives a good approximation of $g(H)$. CProgol4.2 (see Section 5.1) uses an implementation of SLPs in this way for learning from positive data.

The implementations of information content and conditioning (Sections 5.2 and 5.3) are both unsatisfactory in that they assume there is only one derivation of every atom. A complete implementation would sum over all derivations. Unfortunately, it is possible that an infinite set of derivations exist for certain atoms. However, since the probabilities associated with such derivations decrease exponentially in the length of the description it may be the case that summing over the short derivations gives good bounds for the complete sum. This question requires further attention.

The author believes that stochastic logic programs provide an important new representation for machine learning and logic programming.

Acknowledgements

Thanks are due to David Haussler of the University of Santa Cruz for providing me with references to the literature of stochastic grammars. Thanks also for useful discussions on the topics in this paper with Donald Michie, John McCarthy, David Page and Ashwin Srinivasan. This work was supported partly by the Esprit Basic Resaerch Action ILP (6020), the Esprit Long-term Research Action ILP II (LTR 20237), EPSRC grant GR/J46623 on Experimental Application and Development of ILP, EPSRC grant GR/K57985 on Experiments with Distribution-Based Machine Learning and an EPSRC Advanced Research Fellowship held by the author. The author is also supported by a Research Fellowship at Wolfson College Oxford.

References

[1] G. Boole. *The Laws of Thought.* MacMillan & Co., London, 1854.

[2] W. Buntine. *A Theory of Learning Classification Rules.* PhD thesis, School of Computing Science, University of Technology, Sydney, 1990.

[3] R. Carnap. *The logical foundations of probability.* University of Chicago Press, Chicago, 1962.

[4] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[5] A.P. Dempster. A generalisation of bayesian inference. *Journal of the Royal Statistical Society, Series B*, 30:205–247, 1968.

[6] R. Fagin and J. Halpern. Uncertainty, belief and probability. In *Proceedings of IJCAI-89*, San Mateo, CA, 1988. Morgan Kauffman.

[7] D. Haussler, M Kearns, and R. Shapire. Bounds on the sample complexity of Bayesian learning using information theory and the VC dimension. In *COLT-91: Proceedings of the 4th Annual Workshop on Computational Learning Theory*, pages 61–74, San Mateo, CA, 1991. Morgan Kauffmann.

[8] K. Lari and S. J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56, 1990.

[9] J. Lukasiewicz. Logical foundations of probability theory. In L. Berkowski, editor, *Selected works of Jan Lukasiewicz.* North Holland, Amsterdam, 1970.

[10] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[11] S. Muggleton. Bayesian inductive logic programming. In W. Cohen and H. Hirsh, editors, *Proceedings of the Eleventh International Machine Learning Conference*, pages 371–379, San Mateo, CA, 1994. Morgan-Kaufmann.

[12] S. Muggleton. Stochastic logic programs. In L. De Raedt, editor, *Advances in Inductive Logic Programming*. IOS Press/Ohmsha, 1995.

[13] S. Muggleton. Learning from positive data. In *Proceedings of the Sixth Inductive Logic Programming Workshop*, Stockholm University, 1996.

[14] S. Muggleton and C.D. Page. A learnability model for universal representations. Technical Report PRG-TR-3-94, Oxford University Computing Laboratory, Oxford, 1994.

[15] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[16] R. Ng and V.S. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, 1992.

[17] N. Nilsson. Probabilistic logic. *Artificial Intelligence Journal*, 28:71–87, 1986.

[18] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[19] T. Sato. A statistical learning method for logic programs with distributional semantics. In L. Sterling, editor, *Proceedings of the Twelth International conference on logic programming*, pages 715–729, Cambridge, Massachusetts, 1995. MIT Press.

[20] G. Shafer. *A mathematical theory of evidence*. Princeton University Press, Princeton, NJ, 1976.

[21] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.